3. Memory Organization

This section presents details of physical and logical addresses, illustrates the TMS34010 memory map, and describes stack operation.

3.1 Memory Addressing

The TMS34010 is a bit-addressable machine with a 32-bit internal memory address. Each 32-bit address points to an individual bit within memory. Figure 3-1 shows the logical memory structure. Memory is accessed as a continuously addressable string of bits. Groups of adjacent bits form data structures called *fields* (see Section 4). The GSP supports field lengths from 1 to 32 bits. The total memory capacity is four gigabits (or 512 megabytes); the TMS34010 supports external addressing of 128 megabytes. Bit addresses range from >0000 0000 to >FFFF FFFF.

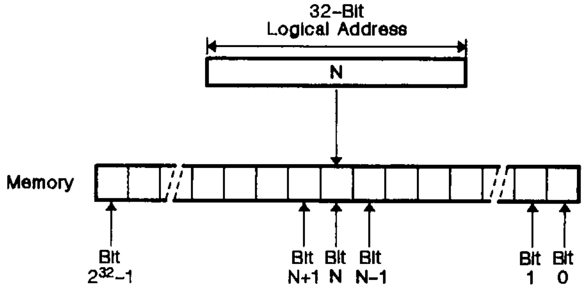


Figure 3-1. Logical Memory Address Space

Figure 3-2 shows the physical memory organization. The GSP communicates with memory over a 16-bit data bus, and always reads or writes a complete 16-bit word from or to memory. The word accessed during a memory cycle always begins on an even 16-bit boundary. That is, the four LSBs of the 32-bit starting address of the word are 0s. Bits within a word are numbered from 0 to 15; bit 15 is the MSB and bit 0 is the LSB. A word is identified by the address of its LSB. In this document, the LSB of a memory word will be depicted as the rightmost bit in the word.

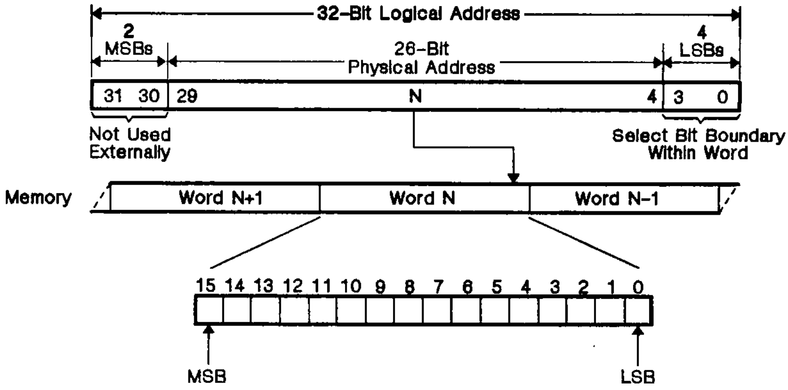


Figure 3-2. Physical Memory Addressing

The four LSBs of the 32-bit logical address in Figure 3-2 do not appear on the local memory bus. When the GSP extracts data that does not begin and end on even word boundaries these four LSBs are used internally to indicate a bit boundary within an accessed word. Control logic at the local memory interface automatically performs the bit alignment and masking necessary to extract the data structure from physical memory. This is completely transparent to software. If the data structure being extracted straddles the boundary between two or more words, multiple read cycles are required. Similarly, inserting a data structure into memory may require a series of read and write cycles, accompanied again by the internal masking and shifting of data to properly align the data structure within memory.

The two MSBs of the 32-bit logical address are not output. The TMS34010 supports an external address range of 128 megabytes of physical memory.

3.2 Memory Map

Figure 3-3 shows the TMS34010 memory map. The memory is divided into three regions:

- Trap vectors
- I/O registers (on chip)
- General use

Memory is logically organized as four gigabits, but is physically accessed 16 bits at a time. Locations are shown as 16-bit words, identified by 32-bit addresses whose four LSBs are 0s. Word addresses range from >0000 0000 to >FFFF FFF0. Bit address >0000 0000 is the rightmost bit in the word at the bottom of the map; bit address >FFFF FFFF is the leftmost bit in the word at the top of the map.

Reading or writing to an address in the range >C000 0000 to >C000 01F0 accesses an internal I/O register. Reading or writing to any address outside this range accesses off-chip memory (or a memory-mapped device) external to the TMS34010.

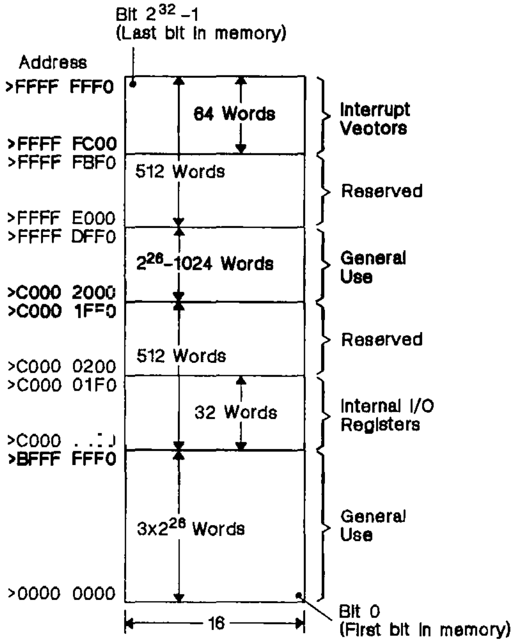


Figure 3-3. TMS34010 Memory Map

Addresses >FFFF FC00 through >FFFF FFE0 are reserved for 32 interrupt, reset, and trap vectors. A vector is a 32-bit address that points to the starting location in memory of the appropriate interrupt, reset, or trap service routine. Each address is stored in physical memory as two consecutive 16-bit words, with the 16 LSBs at the lower address.

The 480 words from >C000 0200 to >C000 1FF0 are reserved for future expansion of the I/O registers. The 448 words from >FFFF E000 to >FFFF FBF0 are reserved for future expansion of the interrupt vectors.

3.3 Stacks

The TMS34010's system stack is implemented in local memory via a dedicated stack pointer (SP) register. The system stack is managed in hardware, and is used to store return addresses and processor status information during interrupts, traps, and subroutine calls. The contents of selected registers in the A and B files can be pushed onto the stack and popped off the stack. The system stack area can also be used for dynamically allocated data storage. The SP is a dedicated 32-bit internal register that points to the top of the system stack. The SP can be accessed by instructions that manipulate registers in either register file.

In addition to the system stack, one or more auxiliary stacks can be managed in software. The system stack always grows toward lower memory addresses, while the auxiliary stack can be defined to grow toward either lower or higher addresses. The MOVE and MOVB instructions, combined with the automatic predecrement and postincrement addressing modes, facilitate pushing and popping auxiliary stack data. One or more registers in the A or B files can be used by software as auxiliary stack pointers and frame pointers. The indexed addressing modes can be used in conjunction with a frame pointer to access variables embedded within the stack.

3.3.1 System Stack

Figure 3-4 shows the structure of the system stack, which grows in the direction of lower memory addresses. The SP points to the top of the stack. That is, it contains the 32-bit address of the LSB (bit 0) of the value on top of the stack. The SP may contain any 32-bit value; however, stack operations execute more efficiently when the four LSBs of the SP are 0s. This aligns the SP to word boundaries in memory, reducing the number of memory cycles necessary to push values onto the stack or pop values off the stack.

During an interrupt, the PC (program counter) and ST (status register) are pushed onto the stack. Instructions that push values onto the system stack include:

- MMTM SP,<register list>
- CALL RS
- CALLA <absolute address>
- CALLR <relative address>
- TRAP <number>
- PUSHST
- MOVE RS,-*SP

Instructions that pop values off the system stack include:

- MMFM SP,<register list>
- RETI
- RETS
- POPST
- MOVE *SP+,RD

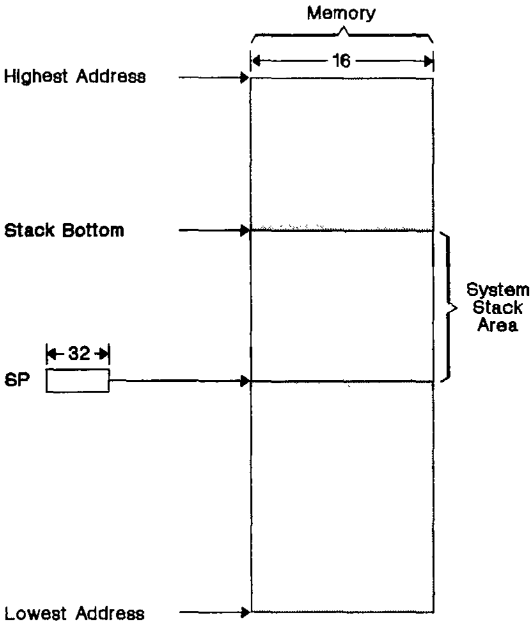


Figure 3-4. System Stack

From one to 16 registers in the A or B file can be moved to or from the stack in a single instruction. The MMTM instruction may be used to push multiple registers onto the stack, and the MMFM instruction may be used to pop multiple registers from the stack. The second word of either instruction is a 16-bit mask, generated from a register list, that specifies which registers in the A or B file are being moved.

The SP can be specified as the source or destination operand in any instruction that operates on the general-purpose registers. Instructions that manipulate registers in the A file or B file can also be used to manipulate the SP.

The contents of 32-bit registers pushed onto the stack are stored in two consecutive words, with the 16 MSBs at the higher memory address, and the 16 LSBs at the lower address. This is shown in Figure 3-5, which demonstrates the effects of the following instruction sequence:

```
MMTM SP,A0        Push register A0 onto stack
MMFM SP,A1        Pop stack into register A1
```

The original state of the stack and registers is shown in Figure 3-5 *a*. Figure 3-5 *b* illustrates the state after A0 has been pushed onto the stack, and Figure 3-5 *c* shows the results of popping the top of the stack into A1.

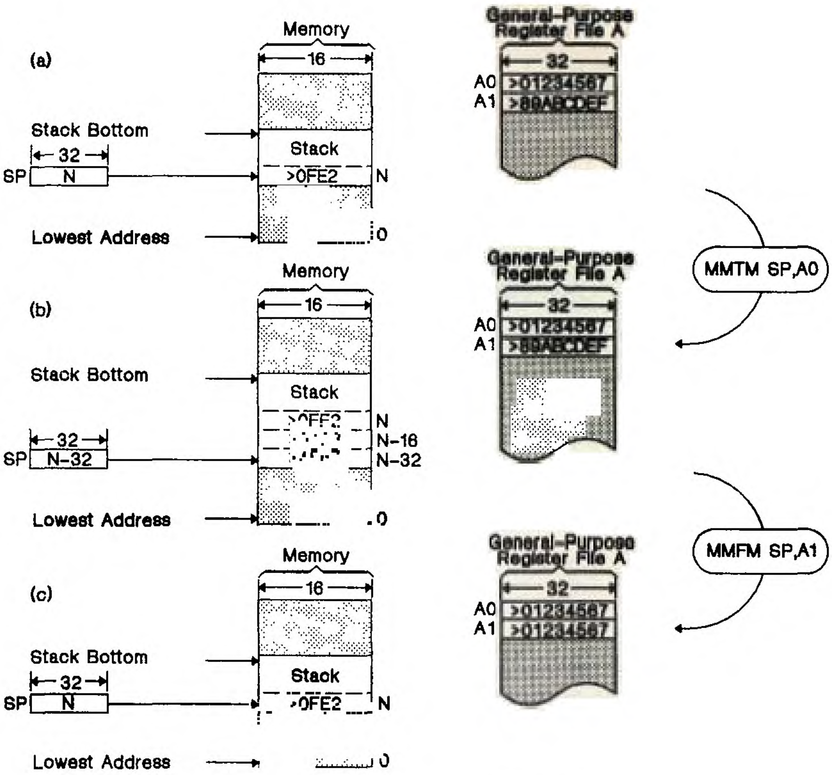


Figure 3-5. Stack Operations

The GSP pushes the contents of a 32-bit register onto the top of the stack according to the following sequence of events:

1) The SP is decremented by 32.
2) The register is pushed onto the stack.

The GSP pops the top of stack into a 32-bit register according to the following sequence of events:

1) The 32 bits at the top of the stack are popped into the register.
2) The SP is incremented by 32.

During an interrupt, the PC and ST are pushed onto the stack to permit the interrupted routine to resume execution when the interrupt processing is completed. The following actions occur during an interrupt routine:

1) The SP is decremented by 32.
2) The PC is pushed onto the stack.
3) The SP is again decremented by 32.
4) The ST is pushed onto the stack.

During a return from an interrupt:

1) The 32 bits at the top of the stack are popped into the ST.
2) The SP is incremented by 32.
3) The 32 bits at the top of the stack are popped into the register.
4) The SP is again incremented by 32.

A subroutine call saves the state of the calling routine on the stack to allow the routine to resume execution when subroutine execution is completed. During a subroutine call, the following actions are taken:

1) The SP is decremented by 32.
2) The PC is pushed onto the stack.

During a return from a subroutine,

1) The 32 bits at the top of the stack are popped into the PC.
2) The SP is incremented by 32.

3.3.2 Auxiliary Stacks

Auxiliary stacks may be managed in software. Any A- or B-file register, except the SP, may be used as the auxiliary stack pointer. Auxiliary stacks are typically used to contain dynamically allocated data storage.

In the following discussion, the symbol *STK* denotes the auxiliary stack pointer. The STK may contain any 32-bit value; however, stack operations execute more efficiently when the four LSBs of the STK are 0s. This aligns the STK to word boundaries in memory, reducing the number of memory cycles necessary to push values onto the stack or pop values off the stack.

As shown in Figure 3-6 and Figure 3-7, the auxiliary stack can be configured to grow in either direction in memory. The memory is shown in these figures as a large set of continuously addressable bits (ignoring for the moment the fact that memory is physically organized as 16-bit words).

The stack shown in Figure 3-6 grows toward lower memory addresses. The original stack is shown in Figure 3-6 *a*. In *b*, a field of arbitrary size is pushed onto the stack via a `MOVE RS,*-STK` instruction (where RS and STK represent general-purpose registers). The field is popped off the stack by a `MOVE *STK+,RD` instruction in *c*. Between instructions, the STK always points to the lowest bit address in the stack – this corresponds to the very top of the stack.

The `MMTM STK,<register list>` instruction can be used to save multiple registers on the stack in Figure 3-6. Later, the registers can be restored to their former values by means of an `MMFM STK,<register list>` instruction.

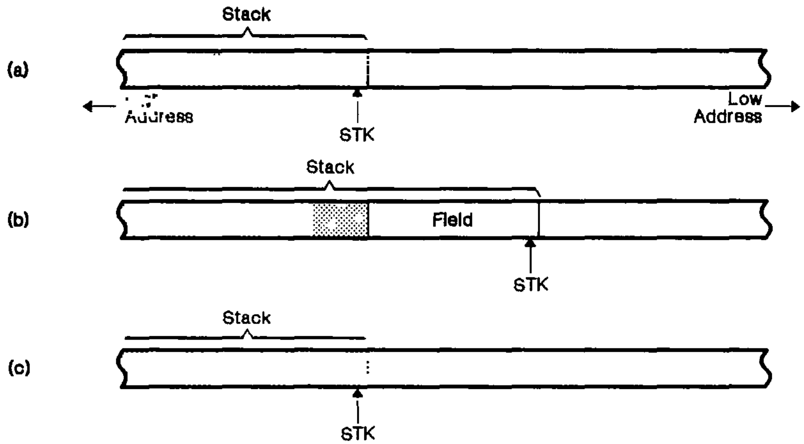


Figure 3-6. Auxiliary Stack Grows toward Lower Addresses

The stack shown in Figure 3-7 grows toward higher memory addresses. The original stack is shown in Figure 3-7 *a*. In *b*, a field of arbitrary size is pushed onto the stack via a `MOVE RS,*STK+` instruction, and in *c* the field is popped off the stack by a `MOVE *-STK,RD` instruction. Between instructions, the STK always points to one plus the highest bit address in the stack – this location is one bit beyond the very top of the stack.

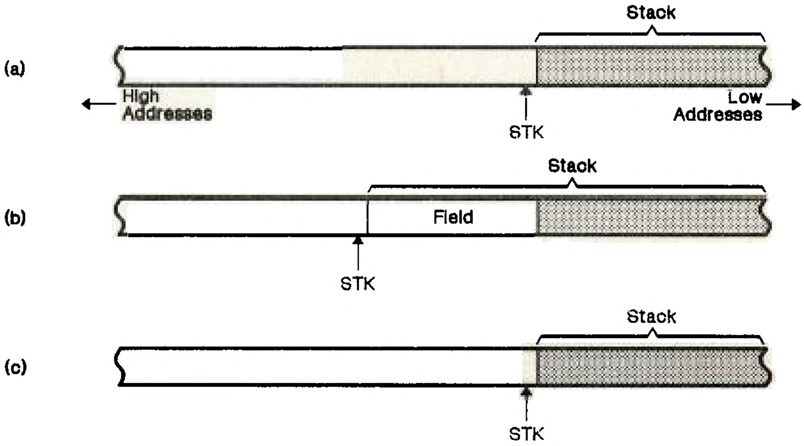


Figure 3-7. Auxiliary Stack Grows toward Higher Addresses